Published in Proc. SC 2002

Asserting Performance Expectations

Jeffrey S. Vetter
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
vetter3@llnl.gov

Patrick H. Worley
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367
worleyph@ornl.gov

Traditional techniques for performance analysis provide a means for extracting and analyzing raw performance information from applications. Users then compare this raw data to their performance expectations for application constructs. This comparison can be tedious for the scale of today's architectures and software systems. To address this situation, we present a methodology and prototype that allows users to assert performance expectations explicitly in their source code using performance assertions. As the application executes, each performance assertion in the application collects data implicitly to verify the assertion. By allowing the user to specify a performance expectation with individual code segments, the runtime system can jettison raw data for measurements that pass their expectation, while reacting to failures with a variety of responses. We present several compelling uses of performance assertions with our operational prototype, including raising a performance exception, validating a performance model, and adapting an algorithm empirically at runtime.

1 Introduction

Traditional techniques for performance analysis provide a variety of mechanisms for instrumentation, data collection, and analysis. These techniques, such as tracing communication activity, sampling hardware counters, and profiling subroutines, allow users to capture raw data about the performance behaviors of their code. Users can then reason about and compare this raw data to their performance expectations for individual application constructs. In most cases, these conventional techniques do not explicitly provide the capability for a user to plant the expectation directly in his or her application. This shortcoming forces users to reason from the perspective of absolute performance for every performance experiment and every application construct. For the scale and complexity of today's architectures and software systems, the volume of raw output can easily overwhelm any user. This comparison can be tedious, difficult, and error-prone.

To address this issue, we present a new methodology and prototype system, called *performance assertions* (PA), that provides this capability so that users can assert performance properties for code constructs explicitly within their applications. The PA runtime implicitly gathers performance data based on the user's assertion and verifies this expectation at runtime. By allowing the user to specify a performance expectation for individual code segments, the runtime system can jettison raw data for successful assertions while reacting to *failures* with a variety of responses. Very simply, this approach attempts to automate the testing of performance properties of evolving complex software systems and software performance models.

To this end, we have implemented an operational prototype for performance assertions. Our experience with this prototype on several applications and with a variety of response mechanisms indicates that performance assertions can improve the traditional process of performance analysis. By providing a higher level of abstraction in the performance analysis process, we permit the user to reason in performance models rather than in the low-level details of instrumentation and data management. That said, we are continuing to improve our prototype based on several observations from these experiments. Key among these observations is the fact that users will need analytical support in determining the bounds for performance assertion expressions. Also, our initial prototype considers only serial performance metrics focused on one processor. We plan to extend this set of metrics in the prototype to include communication, threading, I/O activity, and a combination of these metrics.

1.1 Motivating Example

Traditionally, performance measurement and monitoring has been a multipart process. First, users instrument their applications to capture performance data of interest, which can present a challenge per se because a user must know what data to collect and when to collect it. Users must balance these questions among the competing goals of low overhead, reasonable data volume, and sufficient levels of detail. Next, the users generate performance data from one or more experiments. Third, users analyze this raw data with visualizations or automated tools in the hope of determining if the performance of individual constructs satisfies their expectations. Lastly, with this information in hand, users attempt to optimize constructs that failed their expectations and begin the process anew.

```
PAPI_start(CYCLES,INSTRUCTIONS);
for (j = 1; j <= lastrow - firstrow + 1; j++)
{
  sum = 0.0;
  for (k = rowstr[j]; k < rowstr[j + 1];
       k++)
    {
      sum = sum + a[k] * p[colidx[k]];
    }
  w[j] = sum;
}
PAPI_stop(vals);
/* Analyze or store PAPI values */
```

Figure 1: Traditional instrumentation for a loop.

```
pa_start (&pa, '$ipc_peak*0.5<$ipc');
for (j = 1; j <= lastrow - firstrow + 1; j++)
{
  sum = 0.0;
  for (k = rowstr[j]; k < rowstr[j + 1];
       k++)
    {
      sum = sum + a[k] * p[colidx[k]];
    }
  w[j] = sum;
}
pa_end(pa);
```

Figure 2: Specifying a performance assertion for a loop.

For example, Figure 1 shows a sparse matrix vector multiply (SMVM) loop. To analyze the performance of this loop for instructions per cycle (IPC), users have several options. In this example, we use the Performance Application Programming Interface (PAPI) library [4] to access the underlying hardware counters on the target system. This library returns raw hardware counter values for bracketed regions of code. With each set of values returned by the PAPI library, the application must either store this data for post-mortem analysis or analyze it immediately at runtime. In this example, the instrumentation does not contain any notion of how the data is to be used, so the monitoring system must conservatively record all raw data. PAPI promotes portability for the actual instrumentation process, but it does not address data management and performance expectation issues.

In contrast, Figure 2 shows the same loop when annotated with performance expectations using our performance assertions. By introducing this higher level of abstraction into the performance analysis process, we achieve several goals. In this example, the measurement and data collection mechanisms are no longer pertinent because the PA runtime selects the appropriate instrumentation based on the PA expression and, for example, some platforms use statistical techniques to estimate these hardware values, such as Compaq's DCPI [1]. Second, the PAs can be disabled or removed easily. Third, as the PA is evaluated, the runtime system can purge raw data, keeping only statistics and counts. Fourth, a compiler that recognizes PAs could optimize the PA expression evaluation and minimize overhead due to instrumentation.

In summary, the overall goal of this implementation is to create a source code annotation system for applications that allows a user to specify a performance expectation for a given code segment. At runtime, the assertion will measure the necessary metrics, compare them to the expectation, and, if violated, take some action (e.g., alert the user, enable performance monitoring, adapt the current system). Performance assertions perform three critical tasks. First, they allow the user to define a portable performance expectation in the context of their application design while freeing them from focusing on instrumentation. Second, PAs limit the amount of data that users must encounter during the performance analysis process. By highlighting only those portions of the code that *fail* to meet the user-defined expectation, PAs can preempt data generation before it is thrust upon the user. Third, PAs compel users to express their expectations quantitatively with an expression that reflects their application design, while liberating them from specific instrumentation and portability concerns.

2 Design of Performance Assertions

The design of performance assertions has three distinct components: a performance assertion language, source code annotations, and a runtime system. As illustrated by Figure 3, at step ①, a user annotates source code with performance assertions using the PA language. Next, at step ②, the user executes the annotated source code, and during this execution the PA runtime system collects performance data with instrumentation and evaluates the performance expectations. Finally, at step ③, assertions generate a variety of responses. Assertions that pass can simply be ignored, while failures can trigger an array of responses. For example, in ③a, the final PA report for the application indicates that the assertion failed 13 of 700 invocations.

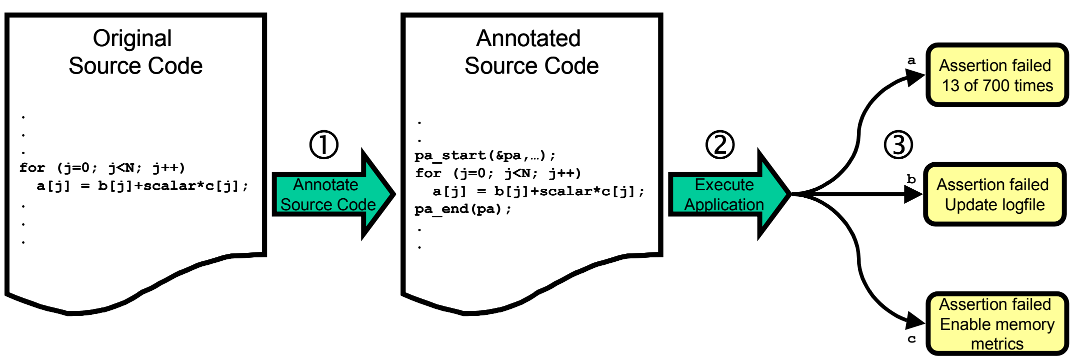

Figure 3: Performance Assertion Overview.

The users define their expectations in our PA language with specific source code annotations as Figure 3 shows; this language provides access to various performance metrics as well as key features of the architecture and user parameters. That is, expressions can contain references to values such as `$wtime` (wall clock time), `$nLoads` (number of memory load instructions), `$nFlops` (number of floating point operations), `$nL1LoadMisses` (number of L1 data cache misses), `$nCyclesReadStall` (cycles stalled on read memory accesses), or `$nInsts` (number of instructions). The PA runtime invokes the proper instrumentation and data collection facilities for each expression. PAs can also reference values that represent architecture characteristics, such as `$fp_peak_rate` (theoretical floating point peak rate), and arbitrary application values can be parameterized into the expression using format specifications similar to `scanf` [8].

The runtime system captures the appropriate metrics and evaluates expressions as necessary, responding with the appropriate action when an assertion fails. The response can take a number of forms. For instance, it can increment a counter, make a callback to a user-defined subroutine, write the data to a log file, or drive feedback into the application or a separate runtime system.

2.1 Performance Assertion Language

Our PA language allows a user to specify an expression that contains a variety of tokens that represent empirically measured performance metrics, constants, variables, mathematical operations, a subset of intrinsic operations (e.g., `log`, `exp`), and format specifiers. Format specifiers allow the expressions to incorporate values from the application directly.

Consider the following example expressions:

`$nInsts / $nCycles > 0.8` (1)

Expression (1) has five tokens. The left-hand side (LHS) of this expression specifies the ratio of number of instructions completed to the number of cycles. The relational operator tests whether the LHS is greater than the right-hand side (RHS), and in this case, the constant `0.8`. When this expression is first evaluated by the PA runtime system, it determines that the underlying instrumentation must collect two performance metrics: number of instructions completed (`nInsts`) and number of cycles (`nCycles`). Subsequent invocations read these metrics from the instrumentation, instantiate the expression's variables, and evaluate the expression.

`$nInsts / $nCycles > (0.4 * $ipc_peak)` (2)

Expression (2) is very similar to expression (1); however, the RHS has been replaced by another expression that contains an architecture-dependent constant: `$ipc_peak`. In order to provide portable, architecture-independent parameterized expressions in our PA language, we have included an array of predefined constants that demonstrate the performance of the underlying architecture. The value for `$ipc_peak` is substituted into the expression at runtime. These constants can be theoretical or empirically measured values, such as those generated with microbenchmarks [9, 10] or machine signatures [14]. These constants are loaded at initialization and they remain constant throughout the application execution.

`$nInsts / $nCycles > (%g * $ipc_peak) , &x` (3)

Expression (3) is very similar to expression (2); however, the RHS has been parameterized to include values directly from the application with the format specifier `%g` and the variable address `&x`. This capability allows users to specialize expressions for specific parameters, such as the size of the input.

Aside from expressiveness, our design of this performance assertion language has several goals, and we attempt to strike a practical balance among these requirements. First, our language must have a flexible, architecture-independent syntax that allows users to express a performance expectation for a component of their source code. With this syntax, users can meld the performance properties in a statement that identifies their expectations for common language and library constructs (e.g., loops, BLAS, or MPI). Second, the language should be relatively simple to interpret, implement, and validate. Because the PA runtime must evaluate the expressions at runtime, it is important that the interpretation and implementation be efficient to minimize PA overhead on the application. (We are investigating dynamic code generation as a potential solution to this issue.) Third, as the earlier examples demonstrate, we need expressive power to allow users to capture complex and important performance characteristics of their applications. We expect the need for complex expressions to grow as users gain more experience with assertions and as the number of performance metrics increases.

Although our current prototype is realized as a library, our language specification is not dependent on our implementation; we plan to integrate performance assertions with a compiler so that PAs can easily benefit from the extensive semantic knowledge of the source code. Indeed, compilers might insert performance assertions automatically to aid in profile-directed compilation [3, 12].

Another benefit of a language specification of performance properties is the opportunity for optimization of the assertion expressions. We consider them portable and flexible because they allow the performance monitoring system to select the appropriate instrumentation and collection mechanisms. For example, two approaches to gathering hardware metrics are sampling and counting. With performance assertions, the runtime system can select the appropriate strategy based on the requirements of the expression. Furthermore, the language can be optimized for the underlying monitoring system on the target architecture, which is similar to Snodgrass' work [15]. Although our language is not as general as a relational query language, it does offer many opportunities for similar optimizations.

Our current implementation relies on source code annotations in the form of library calls to construct and evaluate performance assertions for specific code segments. As mentioned earlier, tight integration with a compiler might pay large dividends by allowing optimization and automatic insertion of these assertions. Currently, the annotations delimit a code segment and an expression as Figure 2 shows.

2.2 Runtime System

In conjunction with source code annotations, our initial implementation of performance assertions uses a runtime system to define assertions, delineate code regions, enable instrumentation, collect data, evaluate expressions, and react to assertion results.

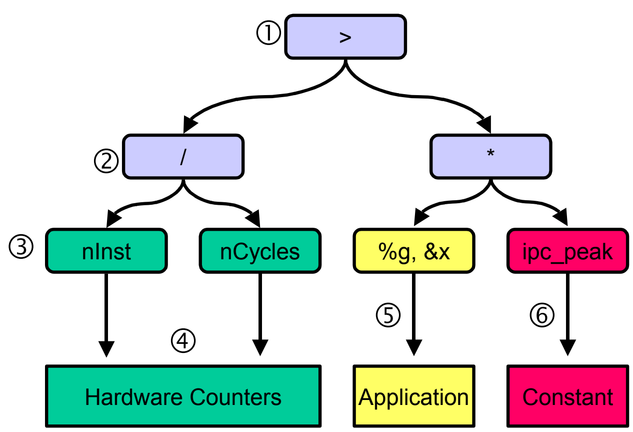

Figure 4: Example expression tree.

As the application encounters PA annotations for the first time at `pa_start`, the subroutine calls the PA runtime to take several steps to initialize the assertion. During initialization, the PA runtime allocates and initializes memory for data storage, parses the expression to determine which tokens represent performance metrics, creates a metric register file that indicates which metrics the assertion must measure during every invocation, and configures any necessary instrumentation. The runtime system parses the expression to determine the necessary performance metrics to gather. Consider parsing expression (3). Figure 4 illustrates the resulting parse tree. The operators in steps ① and ② work as they do in C. The terminals in step ③ illustrate the diversity of data sources for PA expressions. At initialization, the PA runtime scans this parse tree to determine that it must capture two metrics from hardware counters, namely `nInsts` (number of instructions) and `nCycles` (number of cycles) at step ④. The runtime stores this information in the metric register file in the PA handle for this assertion. The runtime, then, examines the relational operator and the RHS of the assertion expression to determine that at step ⑤ the runtime must gather a value the size of a `double` (`%g`) from the address pointed to by `&x`. The runtime reads this value every time the expression is evaluated, so it may change as the application parameters change. Finally, the runtime parses the constant (`ipc_peak`) at step ⑥ and recognizes that it is a constant already defined in its internal symbol table. The runtime stores all this information in the PA handle. At the end of initialization, the PA runtime enables instrumentation.

When the application encounters the `pa_end` call, it reads the metric register file to determine which instrumentation to read and disable. It reevaluates the expression, but this time, it substitutes the actual data values for each terminal into the expression and generates a result. Reconsider our example in Figure 4. Prior to the expression evaluation, the PA runtime has read the current values for `nCycles` and `nInsts` from the hardware counters and has updated its internal metric register file. The values in this register file include an incremental count and an accumulated count. That is, the register file contains the number of cycles since `pa_start` and the accumulated number of cycles for this code region over all previous invocations of this assertion. For the expression evaluation, the PA runtime substitutes the incremental values for `nCycles` and `nInsts` into the calculation. For example, it might find 10,172,045 cycles and 14,136,751 instructions. The runtime, then, computes the result of the division: 1.39. The runtime proceeds to calculate the RHS. It follows the pointer `&x` and retrieves a double value from that address. Next, the runtime extracts the value for `ipc_peak` from the internal symbol table. Assume that the product of this multiplication is 1.25. The runtime compares these two values—1.39 to 1.25—using the greater-than relational operator to discover that the expression is true. Therefore, the assertion is successful. On success, the runtime increments the invocation count and the success count for this assertion, and no action is taken. If, on the other hand, the expression evaluates to false, the invocation count is incremented, the failure count is incremented, and an action is triggered. The PA runtime provides a variety of responses to assertions that a user can select using the PA definition or an environment variable. The action can be ignored, recorded to a log, trigger more detailed monitoring, invoke a user-defined callback, or activate some corrective action, possibly using an adaptation system like Harmony [7] or Autopilot [13]. PA actions are, by default, counters that accumulate the number of failures for an assertion. To specify one of alternative actions mentioned above, users simply call the `pa_set_action` subroutine with the appropriate parameters after defining the assertion.

Subsequent invocations of an assertion simply enable the necessary instrumentation, collect data from the instrumentation, evaluate the expression, and generate an answer. Furthermore, each assertion captures statistics for the values generated from the expressions. These statistics include minimum, maximum, and an accumulated total of all the LHS values.

Naturally, these annotations are easily disabled both at runtime and at compile time. At runtime, a user can disable PAs by using an environment variable or by using specific

PA subroutine calls in their application. At compile time, a user can disable the annotations by using the CPP preprocessor to replace the PA statements with null statements. Disabling PAs at compile time produces an elided version of the application similar to the original. A promising alternative that we are beginning to investigate is to tightly couple the insertion of performance assertions with compilation, so that the combined system can generate assertions automatically using the additional knowledge that a compiler supplies. In this scenario, we hope to use compiler pragmas to define expressions and actions for application statements.

2.3 Generating Bounds

As part of the U.S. Department of Energy (DOE) Scientific Discovery through Advanced Computation project in performance evaluation (http://perc.nersc.gov), we are developing modeling methods that are useful in determining performance properties of a system and that exploit the additional information acquired from performance assertions. Clearly, one primary component of performance assertions is the ability to judge when an assertion has failed. Our initial work exploits other performance measures such as low-level benchmarks and machine signatures. For example, users could state in an expression that they expect a code segment to perform equivalent to the triad benchmark, which is part of the Stream memory suite [9]. Later, we plan to explore more automated techniques. In one instance, the system generates a performance history for each assertion and then compares the assertion with this statistical history across architectures.

3 Compelling Uses of Performance Assertions

Performance assertions have many compelling uses. First, assertions can highlight performance results that do not met user-modeled expectations. Second, PAs can highlight differences across platforms. Third, PAs can draw attention to regions of code that have changing performance expectations as the algorithms and source code evolve. Fourth, PAs can instantiate performance models on small regions of code, alerting users that their modeling assumptions are invalid. Fifth, PAs can trigger a callback into the application or adaptively select among a variety of implementations based on the PA expression.

3.1 Experiment Platforms

We ran our tests on two IBM SP systems located at Lawrence Livermore National Laboratory. The first machine is composed of sixteen 222 MHz IBM Power3 8-way SMP nodes, totaling 128 CPUs. Each processor has three integer units, two floating-point units, and two load/store units. Its 64 KB L1 cache is 128-way associative with 32 byte cache lines, and L1 uses a round-robin replacement scheme. The L2 cache is 8 MB in size, which is 4-way set associative with its own private cache bus. Each SMP node contains 4GB main memory for a total of 64 GB system memory.

The second machine is composed of 332 Mhz 604e 4-way SMP nodes, totaling 1344 CPUs. Each compute node has a peak performance of 2.656 GigaOPS. The 604e processor has one floating-point unit and one load/store unit. Its 32KB L1 cache is 4-way associative with 32 byte cache lines, and L1 uses an LRU replacement scheme. The processor has a 500KB L2 cache.

3.2 Case I: Raising Performance Exceptions

To illustrate the use of performance assertions, we demonstrate how a user can instantiate performance expectations for a given code segment. Then, when that expectation is violated on a different architecture, the user is immediately notified by PAs.

```
for (j = 1; j <= lastrow-firstrow+1; j++)
{
 int iresidue;
 double sum1, sum2;
 i = rowstr[j];
 iresidue = (rowstr[j+1]-i) % 2;
 sum1 = 0.0;
 sum2 = 0.0;
 if (iresidue == 1)
   sum1 = sum1 + a[i]*p[colidx[i]];
 for (k = i+iresidue; k <= rowstr[j+1]-2;
      k += 2) {
   sum1 = sum1 + a[k]   * p[colidx[k]];
   sum2 = sum2 + a[k+1] * p[colidx[k+1]];
   }
 w[j] = sum1 + sum2;
}
```

Figure 5: Unrolled by 2 version of sparse matrix vector multiply for NAS CG.

Our focus is the NAS Benchmark CG, version 2.3. This benchmark uses a sparse matrix vector multiply (SMVM) as illustrated for the NU version in Figure 1 and the U2 version in Figure 5. Its notorious memory access patterns generally require that on the platform's underlying memory architecture be taken into account when designing efficient implementations. In fact, many versions of SMVM exist, each tuned for individual memory architectures. As developers tune this code segment, they have expectations for this code on each architecture. Currently without PAs, there is no way for a developer to insert his or her performance expectations into the source code. Moreover, the only indication that this code segment is not performing well is overall poor application performance.

SMVM VERSION	POWER2 (604E)	POWER3	POWER4
Not unrolled (NU)	78.43	15.24	6.11
Unrolled by 2 (U2)	84.08	15.20	5.80
Unrolled by 8 (U8)	82.53	15.03	6.00

Table 1: Performance of NAS CG with SMVM versions on example architectures.

In Table 1, our experiments show that the tuned performance of SMVM executes quite differently on three different processors. Assumptions about performance of this code on the PowerPC are not transferable, even though they are in the same processor family. On the Power2, the original SMVM (NU) performs best, while on the Power3 the U8 version performs best, and on the Power4 the U2 version outperforms the others. More strikingly, the performance optimum is different for each processor, even though compiled codes (without processor specific instructions) will execute on all three processors.

Performance assertions help to solve this problem because they allow us to insert our expectations directly into the code. First, we add performance assertions to our code with expectations for the IBM 604e processor and then we migrate the code to the IBM Power 3 processor. These chips have different memory and functional unit structures. Using specific information about the memory systems, a user could construct a specific assertion expression, such as `$nL1LoadMisses/$nFlops`, or they could rely on common performance measures, such as instructions per cycle, or even wall clock time scaled by the number of nonzero terms in the operation, to bind their performance property to the target processor. This flexibility allows users to construct the most appropriate expression for their performance property without regard to the mechanics of instrumentation or data collection. Then, when these assumptions are violated, the assertion raises a performance exception. In this example, we found the expression `$nL1LoadMisses/$nFlops`, a reliable predictor of performance of SMVM versions across our example platforms.

3.3 Case II: Validating Performance Models

High performance software usually contains models of performance. In fact, many libraries record metrics about their performance. For example, the PETSC library [2] allows developers to record the number of floating point operations performed during a computational phase. As shown in Figure 6, PAs can easily validate the model by using underlying instrumentation to check the calculation, even integrating application specific data into the expression.

```
 1:   pa_start(&pa, "$nFlops", PA_AEQ, "11 * %g * %g", &ym, &xm);
 2:   for (j=ys; j<ys+ym; j++) {
 3:     for (i=xs; i<xs+xm; i++) {
 4:       if (i == 0 || j == 0 || i == Mx-1 || j == My-1) {
 5:         f[j][i] = x[j][i];
 6:       } else {
 7:         u       = x[j][i];
 8:         uxx     = (two*u - x[j][i-1] - x[j][i+1])*hydhx;
 9:         uyy     = (two*u - x[j-1][i] - x[j+1][i])*hxdhy;
10:         f[j][i] = uxx + uyy - sc*PetscExpScalar(u);
11:       }
12:     }
13:   }
14:   pa_end(pa);
15:   PetscLogFlops(11*ym*xm);
```

Figure 6: Performance model validation.

As the library evolves over time, it is ported to new architectures and is optimized with new techniques. It is useful to validate these models against empirical data. In this example, the library logs the number of flops performed by the doubly nested for loop with the `PetscLogFlops(11*ym*xm)` subroutine. Performance assertions can help validate this claim. At line 1, the `pa_start` describes the expression and delineates the beginning of the code segment: `pa_start(&pa, "$nFlops", PA_AEQ, "11 * %g * %g", &ym, &xm)`. This routine takes as arguments the expression, a relational operator, and threshold or bounds. The expression in this example is the number of floating point operations, `$nFlops`, performed in the code segment. Next, the expression is compared using the relation operator, `PA_AEQ`, which represents approximately equal, or, in this case, ±10% of the threshold value `(11 * %g * %g)`. At line 14, `pa_end` signals the end of the code segment for the matching `pa_start`. `pa_end` collects all the relevant data, calculates the expression, and compares it to the threshold using the relational operator. If this expression fails, the default action notifies the user in a report at application termination. Once the validation is complete, a user can disable the assertions at runtime with an environment variable, or recompile the application to elide the PA statements with the preprocessor.

3.4 Case III: Local Performance-Based Adaptation

Performance assertions can also change the local application state in response to the outcome of its expression. For example, in our prototype, a PA can invoke a user-defined function that can change the state of the application, or select among several alternative implementations based on testing the performance of the alternatives at runtime. For example, our experiences with a Monte Carlo simulation allow us to alter a variety of the application-defined variables in response to performance conditions [5].

```
pa_t pa_smvm;
int smvm_choice = 1;
pa_start(&pa_smvm,"$nCycles/$nInsts",PAR_MINIMIZE, &smvm_choice, 3 );
switch(smvm_choice)
{
  case 1:
    /* SMVM not unrolled */
    break;

  case 2:
    /* SMVM unrolled by 2 */
    break;

  case 3:
    /* SMVM unrolled by 8 */
    break;
}
pa_end(pa_smvm);
```

Figure 7: Performance-based adaptation example using performance assertions.

Reconsider our example in Case I of multiple versions of SMVM. In this example, the user selects one version of the implementation at compile time. Then, if the performance expectation in not satisfied, PAs can notify the user, who in turn changes the implementation, recompiles the application, and executes the code again. Indeed, in this example, we can easily use PAs to evaluate several different versions of the implementation and then, based on the outcome of the samples, select one implementation for the remainder of the application runtime. To implement this strategy, we modify the code in three ways as Figure 7 shows. First, we separate three versions of the implementation with a conditional statement, using a global variable to select among these versions. Then, we register this variable with the PA runtime system. Finally, we create a PA expression that measures the quantity we are interested in minimizing along with a range of possible choices.

Version	CPI
NU	2.38
U2	2.39
U8	2.32

Table 2: Measured CPI on SMVM implementations.

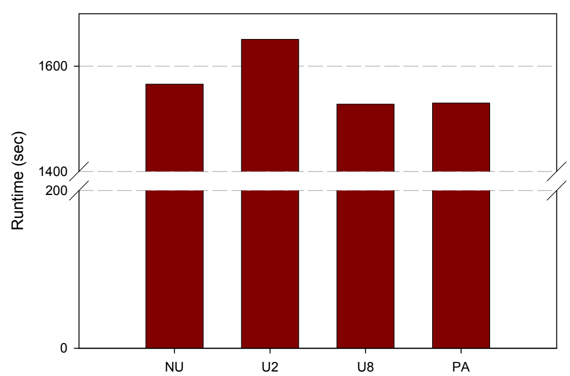

Figure 8: CG performance using various SMVM implementations including PA adaptation.

As the program executes, the PA runtime samples the performance of each implementation using the PA expression as provided by the user. Then, after some number of samples (e.g., in this case, 3 * 20 = 60), it selects one implementation choice by selecting the implementation with the minimal average value of the expression across all samples. Then, this PA disables itself and remains dormant for the remainder of the application execution. Other PAs in this application operate independently. There are a practically innumerable number of ways to adapt the application state in response to PAs.

As Figure 8 shows, our PA adaptation selects the appropriate version of the SMVM on the NAS CG (Class B) benchmark. For this adaptation, we minimized the adaptation expression cycles per instruction. Table 2 supplies the measured CPI for this experiment.

4 Observations

As Section 3 illustrates, performance assertions allow users to insert performance expectations directly into their application code. During our development and testing of the PA system, we made several important observations.

First, we believe that performance assertions are important because they permit users to assert explicitly their performance expectations in their applications. These assertions are then empirically verified at runtime. This is an entirely new way to think about performance analysis. Instead of collecting volumes of performance data and then reasoning about that data on an absolute scale, performance assertions allow users to plant expressions around their important application constructs, and they can elect to be notified only when the assertion is violated at runtime. Although most of our examples focus on the SMVM operation, our intent is to demonstrate that performance assertions are general and can be applied in a variety of situations.

Second, performance assertions depend on the availability, efficiency, and accuracy of the underlying instrumentation. The syntax of performance assertions does not allow a user to specify how performance data is collected; this decision remains with the PA runtime. The benefits of this approach are that the runtime can select the most appropriate method for the instrumentation task at hand and that the performance assertion framework is portable across platforms. Unfortunately, instrumentation support is not easily portable. Tools like PAPI [4] have made great strides in addressing this problem, but microprocessors and operating systems offer a wide range of support for hardware counter instrumentation. Also, the perturbation introduced by performance assertions depends on efficient instrumentation. Our PA runtime does insert a small overhead for parsing the expression, but our measurements show that this overhead scales with the expression complexity, and it typically is a few thousand cycles.

Third, our PA language is expressive to permit users to describe any number of expressions important to them. However, we need to provide additional support to users for understanding how to create these expressions. Although many performance metrics and expressions can identify performance problems, this process of developing and testing these expressions manually can be time consuming and sometimes misleading. Our initial work focuses on using multivariate statistical techniques to correlate individual metrics with overall performance.

5 Related Work

Many research efforts have modeled the performance properties of applications [6, 11]. In fact, the name of performance assertions is not in and of itself novel. However, our technique and prototype, which are novel, allow users to assert explicitly in their code their performance properties, which can be verified empirically at runtime. In contrast to earlier work by Perl [11], this research focuses on runtime techniques to judge if an assertion has met its expectation. Perl's work checked for these properties in event logs, not in the application at runtime. The GrADS project (http://nhse2.cs.rice.edu/grads/) is addressing issues of application performance and performance contracts [17] on computational grids. In a different effort, work by the APART consortium has culminated in a performance property specification language: ASL. ASL allows developers to write complex properties describing patterns in performance data, but current implementations do not allow users to plant their expectations directly in their source code, where they can be measured and verified at runtime. Also, we plan to provide users with a more general framework for reacting to failed assertions [16]. For example, our current work allows assertions to perform local adaptations in response to assertions [5].

6 Conclusions

Traditional techniques for performance analysis provide a means for extracting and analyzing raw performance information from applications. Users then reason about and compare this raw performance data to their performance expectations for important application constructs. This comparison can be tedious, difficult, and error-prone for the scale and complexity of today's architectures and software systems. To address this situation, we present a methodology and prototype that allows users to assert performance expectations explicitly in their source code using performance assertions. As the application executes, each performance assertion in the application collects data implicitly to verify the assertion. By allowing the user to specify a performance expectation with individual code segments, the runtime system can jettison raw data for measurements that pass their expectation, while reacting to failures with a variety of responses. We present several compelling uses of performance assertions with our operational prototype including raising a performance exception, validating a performance model, and adapting an algorithm to an architecture empirically at runtime.

Acknowledgments

The work of Dr. Vetter was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. The work of Dr. Worley was sponsored by the Office of Mathematical, Information, and Computational Sciences, Office of Science, U.S. Department of Energy sponsored this research under Contract No DE-AC05-00OR22725 with UT-Batelle, LLC. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. This paper is available as LLNL Technical Report UCRL-JC-145028.

References

[1] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.-T.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl, “Continuous profiling: where have all the cycles gone?,” *ACM Trans. Computer Systems*, 15(4):357-90, 1997.

[2] S. Balay, W.D. Gropp, L. Curfman McInnes, and B.F. Smith, “Efficient Management of Parallelism in Object Oriented Numerical Software Libraries,” in *Modern Software Tools in Scientific Computing*, E. Arge, A.M. Bruaset et al., Eds.: Birkhauser Press, 1997, pp. 163-202.

[3] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, "ADIFOR - Generating Derivative Codes from Fortran Programs," *Scientific Programming*, 1:1-29, 1992.

[4] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters," Proc. SC2000: High Performance Networking and Computing Conf. (electronic publication), 2000.

[5] I.R. Corey, J.R. Johnson, and J.S. Vetter, "Micro Benchmarking, Performance Assertions and Sensitivity Analysis: A Technique for Developing Adaptive Grid Applications," Proc. Eleventh IEEE International Symp. High Performance Distributed Computing, 2002.

[6] M.E. Crovella and T.J. LeBlanc, "Performance debugging using parallel performance predicates," *SIGPLAN Notices (ACM/ONR Workshop on Parallel and Distributed Debugging)*, 28, no.12:140-50, 1993.

[7] J.K. Hollingsworth and P. Keleher, "Prediction and Adaptation in Active Harmony," Proc. HPDC, 1998, pp. 180-8.

[8] B.W. Kernighan and D.M. Ritchie, *The C programming language*, 2nd ed. Englewood Cliffs, N.J.: Prentice Hall, 1988.

[9] J.D. McCalpin, *Stream Benchmarks*, http://www.cs.virginia.edu/stream, 2002.

[10] P.J. Mucci, K. London, and J. Thurman, "The CacheBench Report," University of Tennessee, Knoxville, TN 1998.

[11] S.E. Perl and W.E. Weihl, "Performance assertion checking," Proc. 14th ACM Symp. Operating Systems Principles, 1993, pp. 134-45.

[12] D. Quinlan, M. Schordan, B. Philip, and M. Kowarschik, "The Specification of Source-To-Source Transformations for the Compile-Time Optimization of Parallel Object-Oriented Scientific Applications," Proc. 14th Workshop on Languages and Compilers for Parallel Computing (LCPC2001), 2001.

[13] R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed, "Autopilot: adaptive control of distributed applications," Proc. Seventh Int'l Symp. High Performance Distributed Computing (HPDC), 1998.

[14] A. Snavely, L. Carrington, and N. Wolter, "Modeling Application Performance by Convolving Machine Signatures with Application Profiles," Proc. IEEE Workshop on Workload Characterization, 2001.

[15] R. Snodgrass, "A Relational Approach to Monitoring Complex Systems," *ACM Trans. Computer Systems*, 6:157-96, 1988.

[16] J.S. Vetter and K. Schwan, "Techniques for delayed binding of monitoring mechanisms to application-specific instrumentation points," Proc. Int'l Conf. Parallel Processing (ICPP), 1998, pp. 477-84.

[17] F. Vraalsen, R.A. Aydt, C.L. Mendes, and D.A. Reed, "Performance Contracts: Predicting and Monitoring Grid Application Behavior," Proc. GRID 2001: Second International Workshop on Grid Computing, 2001, pp. 154-65.